

# Analisis Perbandingan Algoritma Pengurutan (*Sorting*) Berdasarkan Kompleksitas Waktu

Febryan Arota Hia – 13521120<sup>1</sup>  
Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia  
<sup>1</sup>13521120@std.stei.itb.ac.id

**Abstract**—Pada pengolahan suatu data yang berisi kumpulan nilai-nilai, seringkali dibutuhkan sebuah keterurutan di dalamnya. Keterurutan tersebut biasa didapatkan melalui serangkaian proses pengurutan dengan ketentuan tertentu yang disebut algoritma pengurutan (*sorting*). Algoritma pengurutan tersebut sangat penting dalam dunia keinformatika karena dapat mengurangi kompleksitas dari suatu masalah. Algoritma pengurutan juga memiliki pengaplikasian langsung dalam algoritma pencarian, algoritma *database*, algoritma data struktur, dan banyak lagi.

Dalam pengimplementasiannya, algoritma pengurutan memiliki banyak sekali metode, beberapa di antaranya adalah *bubble sort*, *bidirectional bubble sort*, *shaker sort*, *insertion sort*, *selection sort*, *merge sort*, *quick sort*, *heap sort*, *counting sort*, *radix sort*, *bucket sort*, dan lain sebagainya. Dari sekian banyaknya metode pengurutan yang ada, tentu setiap metode tersebut memiliki kelebihan dan kekurangannya masing-masing. Salah satu cara untuk membandingkan kelebihan dari masing-masing metode adalah dengan membandingkan efektivitas atau efisiensi algoritmanya melalui kompleksitas algoritma. Pada makalah ini, analisis kompleksitas pengurutan dibatasi pada 7 metode, yaitu *bubble sort*, *insertion sort*, *selection sort*, *merge sort*, *quick sort*, *heap sort*, dan *shell sort*.

**Keywords**—efisiensi pengurutan, kompleksitas algoritma, pengurutan, *sorting*.

## I. PENDAHULUAN

Algoritma pengurutan adalah algoritma dari serangkaian instruksi yang mengambil larik (*array*) sebagai input lalu melakukan pemrosesan tertentu terhadap larik tersebut kemudian mengeluarkan output berupa larik yang terurut dengan ketentuan tertentu.

Dalam pengimplementasiannya, terdapat berbagai macam metode yang bisa digunakan sebagai algoritma pengurutan. Namun, untuk menentukan metode mana yang paling tepat, perlu dilakukan analisis terlebih dahulu terhadap situasi data yang akan diolah. Masing-masing dari algoritma tersebut memiliki keunikannya masing-masing. Contohnya, beberapa algoritma seperti *merge sort* mungkin akan membutuhkan memori lebih banyak untuk dijalankan dibanding *insertion sort*, meskipun *insertion sort* mungkin tidak secepat *merge sort*.

Algoritma yang efisien adalah algoritma yang meminimumkan kebutuhan waktu serta ruang. Pengukuran kebutuhan waktu dan ruang tersebut dapat dijelaskan melalui kompleksitas waktu dan kompleksitas ruang. Kompleksitas

waktu diukur dari waktu yang dibutuhkan suatu algoritma untuk menyelesaikan eksekusinya. Selanjutnya, kompleksitas ruang adalah kompleksitas yang diukur melalui jumlah total memori yang digunakan suatu algoritma dalam menyelesaikan eksekusinya.

Kedua hal tersebut (kompleksitas waktu dan ruang) memiliki peran penting sebagai parameter untuk mengevaluasi suatu masalah. Namun, dengan adanya kemajuan teknologi saat ini, kompleksitas ruang tidak lagi terlalu diperhatikan karena memori suatu mesin dapat dengan mudah diperbesar. Hal tersebut menjadikan persoalan memori mesin bukanlah hal yang kritis dibandingkan waktu. Maka dari itu, pengevaluasian suatu algoritma lebih sering dianalisis kompleksitas waktunya, pun pada makalah ini hanya akan membahas kompleksitas waktu.

## II. LANDASAN TEORI

### A. Kompleksitas Algoritma

Keefektifan sebuah algoritma dapat dihitung melalui waktu dan ruang yang dibutuhkan untuk mengeksekusi algoritma tersebut. Namun, dengan cara ini, algoritma yang sama pada arsitektur mesin yang berbeda sangat mungkin menghasilkan waktu yang berbeda. Oleh karena itu, dibutuhkan suatu model perhitungan dan pengukuran waktu serta ruang yang independen terhadap mesin. Hal inilah yang disebut dengan kompleksitas algoritma.

Dalam kompleksitas algoritma, terdapat dua macam besaran yang dapat dihitung, yaitu kompleksitas waktu dan ruang. Kompleksitas waktu, biasa disimbolkan  $T(n)$ , merupakan jumlah tahapan komputasi yang dibutuhkan untuk menjalankan suatu algoritma sebagai fungsi dari ukuran masukan  $n$ . Sedangkan kompleksitas ruang, biasa disimbolkan  $S(n)$ , merupakan ukuran dari memori yang dibutuhkan oleh struktur data yang ada pada algoritma sebagai fungsi dari ukuran masukan  $n$ .<sup>[1]</sup>

Kompleksitas waktu digolongkan menjadi 3 macam, yaitu waktu untuk kasus terburuk  $T_{max}(n)$ , waktu untuk kasus rata-rata  $T_{avg}(n)$ , dan waktu untuk kasus terbaik  $T_{min}(n)$ .

Biasanya kompleksitas algoritma dinyatakan secara asimptotik dengan notasi Big-O. Jika kompleksitas waktu untuk

menjalankan suatu algoritma dinyatakan dengan  $T(n)$ , dan memenuhi

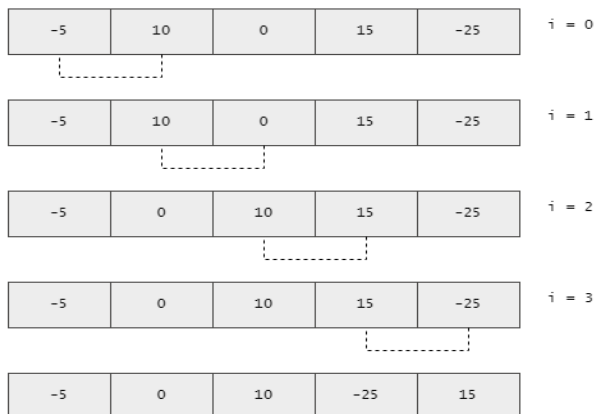
$$T(n) \leq C(f(n))$$

untuk  $n \geq n_0$  maka kompleksitasnya dapat dinyatakan dengan

$$T(n) = O(f(n))$$

### B. Bubble Sort

Algoritma *bubble sort* merupakan proses pengurutan dengan cara membandingkan elemen sebelahnya lalu melakukan pertukaran secara terus menerus hingga dalam satu iterasi tertentu tidak ada lagi perubahan.

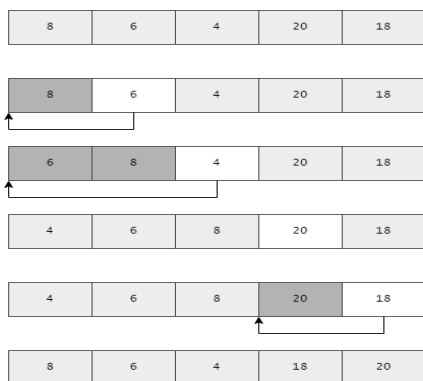


Gambar 1. Skema pengurutan bubble sort  
Sumber: Dokumen penulis

Pada gambar 1 diilustrasikan skema pengurutan membesar (*ascending*), apabila elemen yang dibandingkan lebih besar maka elemen tersebut ditukar. Gambar 1 hanya menggambarkan satu kali iterasi, Pengurutan dilakukan kembali dengan cara yang sama pada iterasi selanjutnya hingga semua elemen terurut.

### C. Insertion Sort

Pengurutan menggunakan *insertion sort* dilakukan dengan cara mengambil elemen satu per-satu dan menyisipkannya pada posisi/urutan yang benar. Penukaran ini dilakukan untuk setiap elemennya hingga seluruh data terurut dengan sesuai.

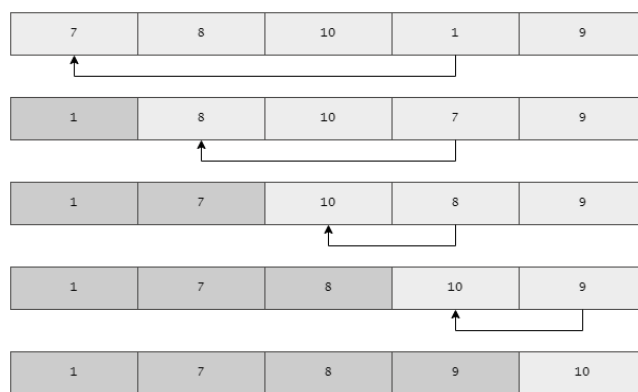


Gambar 2. Skema pengurutan insertion sort  
Sumber: Dokumen penulis

Algoritma *insertion sort* jauh kurang efisien dalam mengolah data dengan jumlah besar apabila dibandingkan dengan algoritma lain seperti *quick sort*, *merge sort* atau *heap sort*. Namun, algoritma jauh lebih efektif untuk kumpulan data yang sudah diurutkan secara substansial dibanding algoritma pengurutan sederhana lainnya seperti *selection sort* atau *bubble sort*.

### D. Selection Sort

Algoritma *selection sort* merupakan metode pengurutan dengan mengambil nilai terbesar atau terkecil dari sekumpulan nilai lalu memindahkannya pada ujung kumpulan nilai tersebut. Algoritma ini memroses dua sub-larik yaitu sub-larik yang sudah terurut, dan sub-larik yang berisi sisa elemen yang belum diproses/diurut. Pada Gambar 3, dapat dilihat sub-larik yang berisi elemen yang sudah diurut berada pada ujung larik sebelah kiri.

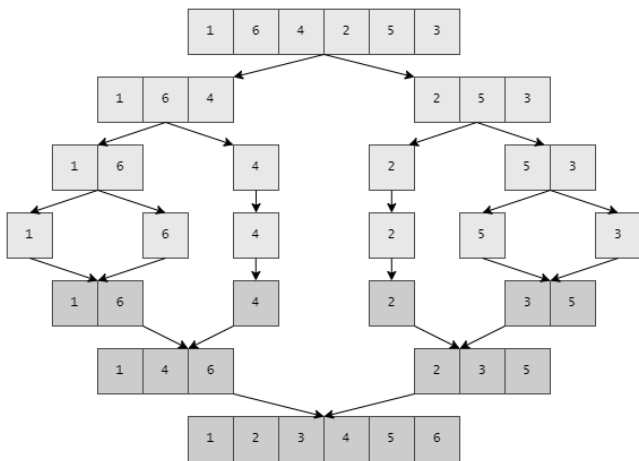


Gambar 3. Skema pengurutan selection sort  
Sumber: Dokumen penulis

Algoritma *selection sort* sering digunakan karena kesederhanaan dalam pengimplementasiannya dibandingkan algoritma-algoritma pengurutan lainnya, Pada gambar 3, langkah pertama dari algoritma ini adalah mencari nilai terkecil yang ada pada sub-larik belum terurut, lalu memindahkannya ke ujung larik sebagai elemen pertama larik yang sudah terurut (digambarkan pada area gelap). Langkah berikutnya, mencari nilai terkecil pada sub-larik belum terurut lalu menempatkannya pada sub-larik terurut sesuai dengan urutan letaknya. Seperti itu seterusnya hingga seluruh elemen larik terurut.

### E. Merge Sort

Algoritma pengurutan *merge sort* dilakukan dengan menggunakan prinsip *divide and conquer* (memecahkan dan menyelesaikan) yaitu dengan membagi data sedemikian rupa kemudian memproses setiap bagian pecahan lalu setelahnya menggabungkan kembali menjadi sebuah data terurut.



Gambar 4. Skema pengurutan merge sort  
Sumber: Dokumen penulis

Pertama, data dibagi menjadi 2 bagian di mana besar masing-masing bagian merupakan setengah dari total jumlah elemen data (apabila elemen genap). Namun, apabila jumlah elemen ganjil, maka bagian pertama menjadi setengah plus satu dari total elemen dan bagian kedua menjadi sisanya. Pembagian atau pemecahan bagian tersebut dilakukan terus menerus hingga masing-masing bagian hanya terdiri dari satu elemen.

Setelah itu masing-masing bagian yang sudah dipecah hingga hanya terdiri dari satu elemen, digabungkan kembali dengan membandingkan bagian yang sama apakah elemen pertama lebih besar dibanding elemen ke-tengah plus satu. Jika lebih besar, maka elemen ke-tengah plus satu dipindah sebagai elemen pertama. Demikian seterusnya hingga menjadi satu bagian utuh seperti yang dapat dilihat pada Gambar 4.

### F. Shell Sort

Algoritma pengurutan *shell sort* melakukan perbandingan suatu data dengan data lain yang memiliki jarak tertentu, kemudian dilakukan penukaran apabila diperlukan. Proses ini dilakukan traversal berkali-kali dan setiap *pass* mengurutkan sejumlah nilai yang sama dengan ukuran set menggunakan *insertion sort*.

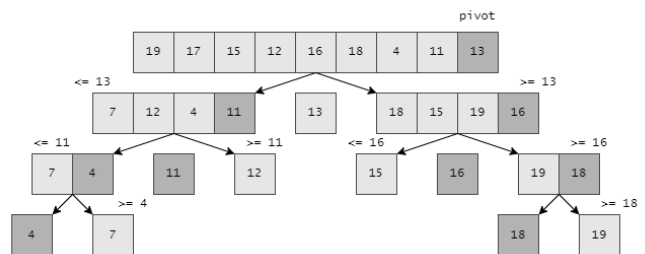
Langkah pertama dalam algoritma pengurutan *shell sort* adalah menentukan jarak mula-mula dari data yang akan dibandingkan, yaitu  $N/2$ . Data pertama dibandingkan dengan data dengan jarak yang sama. Apabila data pertama lebih besar dari data ke  $N/2$  tersebut maka kedua data ditukar. Kemudian data kedua dibandingkan dengan jarak yang sama yaitu  $N/2$ . Demikian seterusnya hingga seluruh data selesai dibandingkan.

Pada langkah berikutnya, digunakan jarak  $\frac{(N/2)}{2}$  atau  $N/4$ . Data pertama dibandingkan dengan data yang memiliki jarak  $N/4$ . Apabila data pertama lebih besar maka kedua data tersebut ditukar. Kemudian data kedua dibandingkan dengan data yang berjarak sama. Langkah tersebut dilakukan seterusnya. Proses berikutnya melakukan hal yang sama, digunakan jarak  $N/8$ , dan seterusnya hingga jarak yang digunakan adalah 1.

Ukuran dari set yang digunakan untuk setiap kali iterasi mempunyai efek besar terhadap efisiensi pengurutan. Tetapi, walaupun tidak seefisien algoritma *merge sort*, *heap sort*, atau *quick sort*, algoritma *shell sort* adalah algoritma yang relatif sederhana. Hal ini menjadikan algoritma *shell sort* adalah pilihan yang baik dan efisien untuk mengurutkan nilai dalam suatu tabel berukuran sedang.

### G. Quick Sort

Algoritma *quick sort* adalah algoritma pengurutan data dengan menggunakan pendekatan rekursif. Proses pengurutan dilakukan dengan membagi atau memecah kumpulan data menjadi dua bagian berdasarkan nilai pivot yang dipilih. Nilai pivot yang dipilih akan diletakkan pada posisi seharusnya di setiap akhir proses pemecahan. Kemudian, setelah seluruh proses pemecahan selesai dan penempatan pivot sudah sesuai, maka proses pengurutan dilanjutkan secara rekursif untuk mengurutkan data bagian kiri dan bagian kanan dari pivot tersebut.



Gambar 5. Skema pengurutan quick sort  
Sumber: Dokumen penulis

### H. Heap Sort

Algoritma *heap sort* adalah algoritma pengurutan berdasarkan perbandingan dan termasuk di dalam golongan *selection sort*. Algoritma pengurutan ini mengurutkan sekumpulan data pada sebuah larik atau *heaptree* (dijelaskan pada paragraf berikutnya). Cara kerjanya adalah, *heap sort* akan mengambil data pada *node* (akar, indeks larik pertama) dan menukarkannya dengan data pada *node* paling akhir. Setelah itu, *node* terakhir dihapus dan *heap sort* meneruskan prosedur *heapify*.

*Heap tree* merupakan *Complete Binary Tree* (CBT) di mana nilai *key* pada tiap *node*-nya sedemikian rupa sehingga *child* di bawahnya tidak ada yang lebih besar dari nilai *key* pada *node parent*.

Berikutnya, algoritma prosedur *heapify* adalah proses melakukan iterasi mulai dari internal *node* paling bawah dan paling kanan hingga *root*, kemudian ke arah kiri dan naik ke level di atasnya, dan seterusnya hingga mencapai *root*. Pada internal *node* tersebut, pemeriksaan hanya dilakukan pada *node* anaknya.

### III. ANALISIS KOMPLEKSITAS ALGORITMA

#### A. Bubble Sort

```
void BubbleSort(int *T, int Nmax) {
    int i, j, temp;
    for (i = (Nmax-1); i >= 0; i--) {
        for (j = 1; j <= i; j++) {
            if(T[j-1] > T[j]) {
                temp = T[j-1];
                T[j-1] = T[j];
                T[j] = temp;
            }
        }
    }
}
```

Gambar 6. Code bubble sort dalam bahasa C  
Sumber: Dokumen penulis

Pada kondisi *Best-Case* (kasus terbaik), yaitu ketika data sudah terurut sebelumnya, proses perbandingan hanya dilakukan sebanyak  $(n - 1)$  kali. Maka, nilai kompleksitas algoritmanya adalah  $O(n)$ .

Pada kondisi *Worst-Case* (kasus terburuk), yaitu ketika data terurut terbalik, maka untuk iterasi pertama akan melakukan  $(n - 1)$  perbandingan, iterasi kedua melakukan  $(n - 2)$  perbandingan, iterasi ketiga melakukan  $(n - 3)$  perbandingan, dan seterusnya. Didapatkan total seluruh perbandingan

$$T(n) = (n - 1) + (n - 2) + \dots + 1$$

$$T(n) = \frac{n(n - 1)}{2}$$

Dari persamaan di atas, didapatkan nilai kompleksitas algoritmanya  $O(n^2)$ . Dari kedua keadaan tersebut dapat dicari juga kompleksitas waktu pada kondisi *Average-Case* yaitu  $O(n^2)$ .

#### B. Insertion Sort

```
void InsertionSort(int T[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++){
        key = T[i];
        j = i - 1;

        while (j >= 0 && T[j] > key) {
            T[j+1] = T[j];
            j--;
        }
        T[j+1] = key;
    }
}
```

Gambar 7. Code insertion sort dalam bahasa C  
Sumber: Dokumen penulis

Pada *Best-Case* skenario, program hanya akan melakukan perbandingan sebanyak  $(n - 1)$  kali tanpa melakukan instruksi pertukaran karena pada kondisi *Best-Case* data telah terurut.

Pada kasus *Worst-Case*, program akan melakukan perbandingan pada iterasi pertama, yaitu membandingkan elemen kedua dengan pertama lalu meletakkannya di tempat yang tepat

(1 kali perbandingan). Untuk iterasi kedua, dibandingkan elemen ketiga dengan dua elemen pertama (2 kali perbandingan). Untuk iterasi ketiga, dibandingkan elemen keempat dengan tiga elemen pertama (3 kali perbandingan). Perbandingan tersebut dilakukan hingga elemen terakhir. Maka, total perbandingan yang dilakukan pada *worst-case* menjadi

$$T(n) = \frac{n(n - 1)}{2}$$

$$T(n) = n^2 - n$$

Dari persamaan di atas, didapatkan nilai kompleksitas algoritmanya  $O(n^2)$ . Dari kedua keadaan tersebut dapat dicari juga kompleksitas waktu pada kondisi *Average-Case* yaitu  $O(n^2)$ .

#### C. Selection Sort

```
void SelectionSort(int T[], int n) {
    int i, j, minIDX;
    for (i = 0; i < n-1; i++) {
        minIDX = i;
        for (j = i+1; j < n; j++) {
            if (T[j] < T[minIDX]) {
                if (minIDX != i) {
                    swap(&T[minIDX], &T[j]);
                }
            }
        }
    }
}
```

Gambar 8. Code selection sort dalam bahasa C  
Sumber: Dokumen penulis

Sama seperti pada algoritma *bubble sort*, *selection sort* akan melakukan sejumlah iterasi dengan iterasi pertama melakukan perbandingan sebanyak  $(n - 1)$  kali perbandingan. Lalu untuk iterasi kedua dilakukan sebanyak  $(n - 2)$  kali perbandingan, dan begitu seterusnya. Maka, didapatkan

$$T(n) = \frac{n(n - 1)}{2}$$

Dari persamaan diatas, didapatkan kompleksitas algoritma *selection sort* senilai  $O(n^2)$ . Dapat dilihat bahwa apapun input/masukan array (belum terurut, terurut terbalik, atau sudah terurut), loop pada Gambar 8 akan dieksekusi secara utuh. Oleh karena itu, kompleksitas waktu *selection sort* pada *best-case*, *worst-case*, ataupun *average-case* akan bernilai sama, yaitu  $O(n^2)$ .

#### D. Merge Sort

```
/* l is for left index and r is right index of the sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - 1) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Gambar 9. Code merge sort dalam bahasa C [2]  
Sumber: Dokumen penulis

```

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    /* create temp arrays */
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    /* Merge the temp arrays back into arr[l..r]*/
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Copy the remaining elements of L[],
    if there are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

```

Gambar 10. Code fungsi merge<sup>[2]</sup>  
 Sumber: Dokumen penulis

Pada algoritma *merge sort* terdapat sejumlah iterasi (*pass*) yang dilakukan untuk memproses input/data. *Pass* pertama menggabungkan bagian atau segmen ukuran 1. Lalu *pass* kedua menggabungkan segmen berukuran 2, begitu seterusnya hingga *pass* ke-*i* menggabungkan segmen berukuran  $2i - 1$ . Apabila dijumlahkan, *pass* tersebut akan memiliki total sebanyak  $\log_2 n$  *pass*.

Dalam penggabungan dua segmen yang telah diurut pada setiap *pass*, memerlukan  $O(n)$ . Sehingga total kompleksitas waktu yang didapatkan sebesar  $O(n \log n)$ . Kompleksitas waktu ini berlaku untuk semua kasus, baik *best-case*, *worst-case*, maupun *average case*.

### E. Shell Sort

```

void ShellSort(int array[], int n) {
    // Rearrange elements at each n/2, n/4, n/8, ... intervals
    for (int interval = n / 2; interval > 0; interval /= 2) {
        for (int i = interval; i < n; i += interval) {
            int temp = array[i];
            int j;
            for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
                array[j] = array[j - interval];
            }
            array[j] = temp;
        }
    }
}

```

Gambar 11. Code shell sort dalam bahasa C<sup>[3]</sup>  
 Sumber: Dokumen penulis

Kompleksitas waktu *shell sort* bergantung pada urutan *gap*. Kompleksitas waktu pada kasus terbaik adalah  $O(n \log n)$  dan kasus terburuknya adalah  $O(n (\log n)^2)$ . Kompleksitas waktu pengurutan *shell sort* umumnya diasumsikan mendekati  $O(n)$  dan kurang dari  $O(n^2)$ .<sup>[3]</sup>

### F. Quick Sort

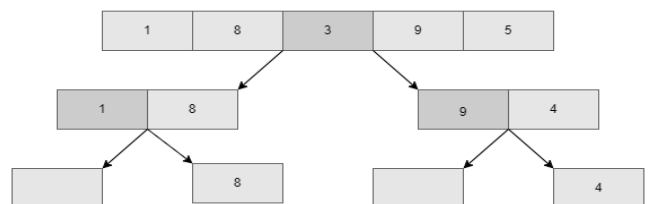
```

void QuickSort(int T[], int Nmax) {
    q_sort(T, 0, Nmax - 1);
}
void Q_sort(int T[], int left, int right) {
    int pivot, l_hold, r_hold;
    l_hold = left;
    r_hold = right;
    pivot = T[left];
    while (left < right) {
        while ((T[right] >= pivot) && (left < right))
            right--;
        if (left != right) {
            T[left] = T[right];
            left++;
        }
        while ((T[left] <= pivot) && (left < right))
            left++;
        if (left != right) {
            T[right] = T[left];
            right--;
        }
    }
    T[left] = pivot;
    pivot = left;
    left = l_hold;
    right = r_hold;
    if (left < pivot)
        q_sort(T, left, pivot-1);
    if (right > pivot)
        q_sort(T, pivot+1, right);
}

```

Gambar 12. Code quick sort dalam bahasa C<sup>[4]</sup>  
 Sumber: Dokumen penulis

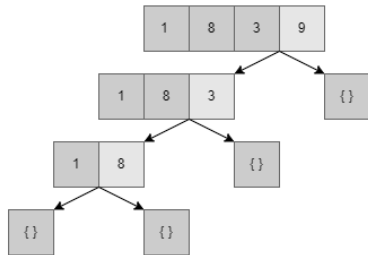
Pada algoritma *quick sort*, kasus terbaik (*best-case*) terjadi ketika elemen yang dipilih sebagai pivot adalah elemen rata-rata dari larik. Dalam kondisi ini tinggi dari pohon adalah  $\log(n)$  dan pada setiap level akan dilakukan pemrosesan ke semua elemen dengan total operasi akan menjadi  $n \log n$ .<sup>[8]</sup> Dari total operasi tersebut didapatkan kompleksitas waktu senilai  $O(n \log n)$ . Pemilihan elemen rata-rata sebagai pivot tersebut membuat pembagian larik menjadi cabang-cabang dengan ukuran yang sama. Oleh karena itu ketinggian pohon menjadi minimum seperti pada Gambar 13.



Gambar 13. Diagram quick sort pada best-case  
 Sumber: Dokumen penulis

Untuk kasus rata-rata atau *average-case*, kompleksitas algoritmanya akan tidak jauh berbeda dan dapat dianggap sama seperti pada kondisi *best-case*, yaitu  $O(n \log n)$ .

Untuk kondisi *worst-case*, terjadi ketika pivot yang dipilih adalah elemen dengan indeks terkecil atau terbesar seperti pada gambar 14. Pemilihan pivot akan selalu dipilih pada indeks terujung sehingga ketinggian pohon adalah  $n$  dan di tiap simpul atas akan melakukan operasi sebanyak  $n$ , lalu untuk simpul berikutnya dilakukan operasi sebanyak  $n - 1$ , demikian seterusnya hingga 1. Berdasarkan kondisi tersebut didapatkan kompleksitas waktu senilai  $O(n^2)$ .



Gambar 14. Diagram quick sort pada *worst-case*  
Sumber: Dokumen penulis

### G. Heap Sort

```
void heapify(int arr[], int N, int i) {
    // Initialize largest as root
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    // If left child is larger than root
    if (left < N && arr[left] > arr[largest]) {
        largest = left;
    }
    // If right child is larger than largest so far
    if (right < N && arr[right] > arr[largest]) {
        largest = right;
    }
    // Swap and continue heapifying if root is not
    //largest If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        // Recursively heapify
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N) {
    // Build max heap
    for (int i = N / 2 - 1; i >= 0; i--) {
        heapify(arr, N, i);
    }
    // Heap sort
    for (int i = N - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```

Gambar 15. Code heap sort dalam bahasa C<sup>[5]</sup>  
Sumber: Dokumen penulis

Algoritma *heap tree* menggunakan pohon biner di mana bagian bawah *heap* berisi jumlah maksimum dari *node*. Ketika menaiki satu level, jumlah *node* berkurang setengahnya. Mengingat terdapat  $n$  jumlah *node*, maka jumlah *node* dari level paling bawah adalah  $n/2$ , lalu  $n/4$  pada level berikutnya, lalu  $n/8$  dan seterusnya.

Oleh karena itu, ketika *heap sort* memasukkan nilai baru ke dalam *heap*, jumlah operasi maksimal yang perlu dilakukan adalah  $O(\log n)$ . Saat memasukkan nilai baru di *heap*, program akan menukarnya dengan nilai yang lebih besar, jumlah penukaran tersebut adalah  $O(\log n)$ . Demikian juga ketika menghapus *node* bernilai maksimal pada *heap*, jumlah operasi yang diperlukan akan menjadi  $O(\log n)$ .

Dapat dilihat bahwa, penghapusan pertama dari sebuah *node* membutuhkan waktu  $\log(n)$ . Penghapusan kedua membutuhkan waktu  $\log(n - 1)$ . Penghapusan ketiga membutuhkan waktu  $\log(n - 2)$ , dan seterusnya hingga penghapusan *node* terakhir yang membutuhkan waktu  $\log(1)$ .

$$T(n) = \log(n) + \log(n - 1) + \log(n - 2) + \dots + \log(1)$$

$$T(n) = \log(n(n - 1)(n - 2) \dots (1))$$

$$T(n) = \log(n!)$$

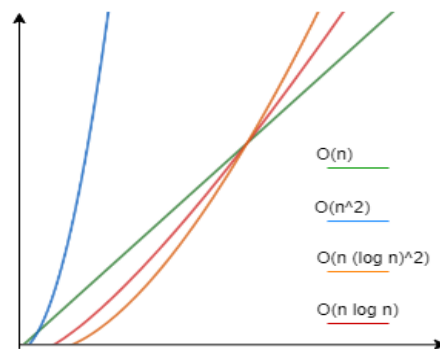
$$T(n) = n \log(n) - n$$

Dari persamaan di atas didapatkan kompleksitas waktu *heap sort*, yaitu  $O(n \log(n))$ . Kompleksitas ini berlaku baik pada kondisi *best-case*, *average-case*, maupun *worst-case*.

### H. Perbandingan Kompleksitas

Algoritma	Kompleksitas Waktu ( <i>case</i> )		
	<i>Best</i>	<i>Worst</i>	<i>Average</i>
<i>Bubble Sort</i>	$O(n)$	$O(n^2)$	$O(n^2)$
<i>Insertion Sort</i>	$O(n)$	$O(n^2)$	$O(n^2)$
<i>Selection Sort</i>	$O(n^2)$	$O(n^2)$	$O(n^2)$
<i>Merge Sort</i>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
<i>Shell Sort</i>	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$
<i>Quick Sort</i>	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
<i>Heap Sort</i>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Tabel 1. Perbandingan kompleksitas



Gambar 16. Grafik kompleksitas waktu  
Sumber: Dokumen penulis



#### IV. KESIMPULAN

Berdasarkan hasil analisis, disimpulkan bahwa dalam proses pengurutan data dapat digunakan algoritma pengurutan sesuai dengan kondisi data yang ada. Pemilihan algoritma tersebut dapat disesuaikan berdasarkan kompleksitas waktunya, seperti yang dapat dilihat pada hasil analisis Tabel 1.

Untuk pengurutan data yang tidak terlalu besar, *quick sort* atau *shell sort* adalah pilihan yang tepat. Sedangkan untuk mengolah data yang besar, dapat digunakan *merge sort* atau *heap sort*.

#### V. UCAPAN TERIMA KASIH

Puji syukur penulis panjatkan kepada Allah SWT karena berkat-Nya, penulis dapat menyelesaikan tugas makalah ini. Dengan segala kerendahan hati, penulis ingin menyampaikan ucapan terima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T. selaku dosen Kelas 02 Mata Kuliah Matematika Diskrit atas segala ilmu dan bimbingannya. Penulis juga ingin mencapaikan ucapan terima kasih kepada orang tua penulis serta teman-teman yang sudah mendukung dalam penyusunan makalah ini.

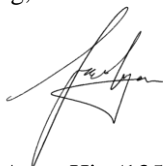
#### REFERENSI

- [1] Munir, Rinaldi “Kompleksitas Algoritma Bagian 1” <https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf> (diakses 8 Desember 2022)
- [2] Geeks for Geeks. “Merge Sort Algorithm”. <https://www.geeksforgeeks.org/merge-sort/> (diakses 8 Desember 2022)
- [3] Programiz. “Shell Sort Algorithm” <https://www.programiz.com/dsa/shell-sort> (diakses 9 Desember 2022)
- [4] Geeks for Geeks. “Heap Sort Algorithm”. <https://www.geeksforgeeks.org/heap-sort/> (diakses 9 Desember 2022)
- [5] Geeks for Geeks. “Heap Sort Algorithm”. <https://www.geeksforgeeks.org/heap-sort/> (diakses 9 Desember 2022)
- [6] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison–Wesley, 1997. ISBN 0-201-89685-0. Pages 138–141 of Section 5.2.3: Sorting by Selection.
- [7] Cici Al Akhyati. “Perangkat Lunak Pendukung Pembelajaran Algoritma Heap Sort).
- [8] Priyansh Mangal. “Shell Sort Algorithm- Explanation, Implementation and Complexity” 2016. <https://www.codingeek.com/algorithms/shell-sort-algorithm-explanation-implementation-and-complexity/#:~:text=The%20time%20complexity%20of%20Shell,is%20still%20an%20open%20problem> (diakses 10 Desember 2022)
- [9] Musliadi. “Memahami Konsep Bubble Sort (Step by Step)”. <https://medium.com/@musliadi/memahami-konsep-bubble-sort-step-by-step-cab7fa85edbe> (diakses 8 Desember 2022)

#### PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 10 Desember 2020



Febryan Arota Hia (13521120)